

Pruebas automáticas

lección3

Las pruebas son la parte más importante del desarrollo de software. No importa cuán bonita, rápida o tecnológicamente avanzada sea tu aplicación, si falla, darás una impresión muy pobre.

Hacer pruebas manuales, es decir, abrir el navegador y ejecutar la aplicación exactamente como lo haría un usuario final, no es viable; porque el problema real no está en el código que acabas de escribir, sino en el código que ya estaba ahí. Normalmente pruebas el código que acabas de escribir, pero no pruebas todo el código que ya existe en tu aplicación. Y sabes muy bien que cuando tocas cualquier parte de tu aplicación puedes romper cualquier otra parte inadvertidamente.

Necesitas poder hacer cualquier cambio en tu código con la tranquilidad de que no vas a romper tu aplicación. Una forma de conseguirlo, es usando pruebas automáticas. Vamos a hacer pruebas automáticas usando JUnit.

3.1 JUnit

JUnit³ es un herramienta muy popular para hacer pruebas automáticas. Esta herramienta está integrada con Eclipse, por tanto no necesitas descargarla para poder usarla. OpenXava extiende las capacidades de JUnit para permitir probar un módulo de OpenXava exactamente de la misma forma que lo haría un usuario final. De hecho, OpenXava usa HtmlUnit⁴, un software que simula un navegador real (incluyendo JavaScript) desde Java. Todo está disponible desde la clase de OpenXava `ModuleTestBase`, que te permite automatizar las pruebas que tú harías a mano usando un navegador de verdad de una forma simple.

La mejor manera de entender como funcionan las pruebas en OpenXava es verlo en acción.

3.2 ModuleTestBase para probar módulos

Para crear una prueba para un módulo de OpenXava extendemos de la clase `ModuleTestBase` del paquete `org.openxava.tests`. Ésta clase te permite conectar con un módulo OpenXava como un navegador real, y tiene muchos métodos útiles para probar tu módulo. Creemos la prueba para tu módulo `Customer`.

3.2.1 El código para la prueba

Creo un paquete nuevo llamado `org.openxava.invoicing.tests` y dentro

³ <http://www.junit.org>

⁴ <http://htmlunit.sourceforge.net>

45 Lección 3: Pruebas automáticas

de él una nueva clase llamada CustomerTest con el código del listado 3.1.

Listado 3.1 Prueba automática JUnit para el módulo Customer

```
package org.openxava.invoicing.tests;

import org.openxava.tests.*;

public class CustomerTest extends ModuleTestBase { // Ha de extender de
                                                    // ModuleTestBase

    public CustomerTest(String testName) {
        super(testName,
            "Invoicing", // Indicamos el nombre de aplicación (Invoicing)
            "Customer"); // y nombre de módulo (Customer)
    }

    public void testCreateReadUpdateDelete() throws Exception { // Los métodos
                                                                // de prueba han de empezar por 'test'

        // Crear
        execute("CRUD.new"); // Pulsa el botón 'New'
        setValue("number", "77"); // Teclea 77 como valor para el campo 'number'
        setValue("name", "JUNIT Customer"); // ← Pon valor en el campo 'name'
        setValue("address.street", "JUNIT Street"); // Fijate en la notación del punto
                                                    // para acceder al miembro de la referencia
        setValue("address.zipCode", "77555"); // Etc
        setValue("address.city", "The JUNIT city"); // Etc
        setValue("address.state", "The JUNIT state"); // Etc

        execute("CRUD.save"); // Pulsa el botón 'Save'
        assertNoErrors(); // Verifica que la aplicación no muestra errores
        assertEquals("number", ""); // Verifica que el campo 'number' está vacío
        assertEquals("name", ""); // Verifica que el campo 'name' está vacío
        assertEquals("address.street", ""); // Etc
        assertEquals("address.zipCode", ""); // Etc
        assertEquals("address.city", ""); // Etc
        assertEquals("address.state", ""); // Etc

        // Leer
        setValue("number", "77"); // Pon 77 como valor para el campo 'number'
        execute("CRUD.refresh"); // Pulsa el botón 'Refresh'
        assertEquals("number", "77"); // Verifica que el campo 'number' tiene un 77
        assertEquals("name", "JUNIT Customer"); // y 'name' tiene 'JUNIT Customer'
        assertEquals("address.street", "JUNIT Street"); // Etc
        assertEquals("address.zipCode", "77555"); // Etc
        assertEquals("address.city", "The JUNIT city"); // Etc
        assertEquals("address.state", "The JUNIT state"); // Etc

        // Actualizar
        setValue("name", "JUNIT Customer MODIFIED"); // Cambia el valor del
                                                    // campo 'name'
        execute("CRUD.save"); // Pulsa el botón 'Save'
        assertNoErrors(); // Verifica que la aplicación no muestra errores
        assertEquals("number", ""); // Verifica que el campo 'number' está vacío
        assertEquals("name", ""); // Verifica que el campo 'name' está vacío

        // Verifica si se ha modificado
        setValue("number", "77"); // Pon 77 como valor para el campo 'number'
        execute("CRUD.refresh"); // Pulsa en el botón 'Refresh'
        assertEquals("number", "77"); // Verifica que el campo 'number' tiene un 77
        assertEquals("name", "JUNIT Customer MODIFIED"); // y 'name'
```

```

// tiene 'JUNIT Customer MODIFIED'

// Borrar
execute("CRUD.delete"); // Pulsa en el botón 'Delete'
assertMessage("Customer deleted successfully"); // Verifica que el mensaje
// 'Customer deleted successfully' se muestra al usuario

}
}

```

Esta prueba crea un nuevo cliente, lo busca, lo modifica y al final lo borra. Aquí ves como puedes usar métodos como `execute()` o `setValue()` para simular las acciones del usuario, y métodos como `assertValue()`, `assertNoErrors()` o `assertMessage()` para verificar el estado de la interfaz de usuario. Tu prueba actúa como las manos y los ojos del usuario. Mira la figura 3.1.

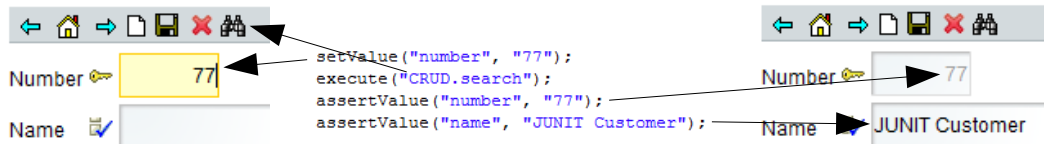


Figura 3.1 Cada línea de la prueba simula una acción del usuario o verifica el estado de la interfaz de usuario

En `execute()` tienes que especificar el nombre calificado de la acción, esto quiere decir `NombreControlador.nombreAcción`. ¿Cómo puede saber el nombre de la acción? Pasea tu ratón sobre el vínculo de la acción, y verás en la barra inferior de tu navegador un código JavaScript que incluye el nombre calificado de la acción. Tal como muestra la figura 3.2.

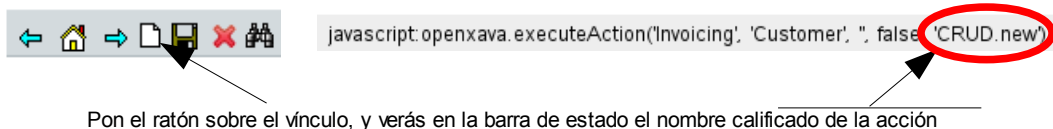


Figura 3.2 La barra inferior del navegador tiene el nombre calificado de la acción

Ahora ya sabes como crear una prueba para probar las operaciones de mantenimiento básicas de un módulo. No es necesario escribir una prueba demasiado exhaustiva al principio. Simplemente prueba las cosas básicas, aquellas cosas que normalmente probarías con un navegador. Tu prueba crecerá de forma natural a medida que tu aplicación crezca y los usuarios vayan encontrando fallos.

Aprendamos como ejecutar tu prueba desde Eclipse.

3.2.2 Ejecutar las pruebas desde Eclipse

JUnit está integrado dentro de Eclipse, por eso ejecutar tus prueba en Eclipse es más fácil que quitarle un caramelo a un niño. Pon el ratón sobre tu clase de prueba, y con el botón derecho escoge *Run As > JUnit Test*. Como muestra la figura 3.3.

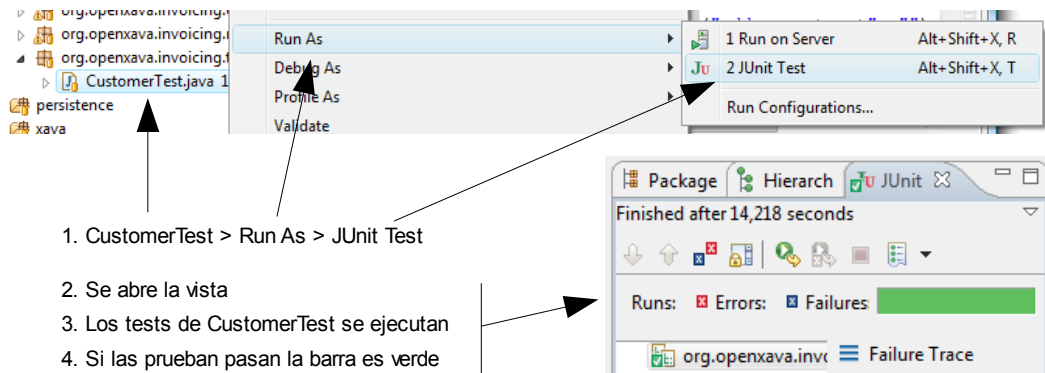


Figura 3.3 Executar pruebas JUnit desde Eclipse

Si la prueba no es satisfactoria la barra sale roja. Puedes probarlo. Edita CustomerTest y comenta la línea que da valor al campo name field. Mira el listado 3.2.

Listado 3.2 Modificar la prueba para que falle

```
...
setValue("number", "77");
// setValue("name", "JUNIT Customer"); // Comenta esta línea
setValue("address.street", "JUNIT Street");
...
```

Ahora, reejecuta la prueba. Ya que name es una propiedad requerida, un mensaje de error será mostrado al usuario, y el objeto no se grabará. Míralo en la figura 3.4.

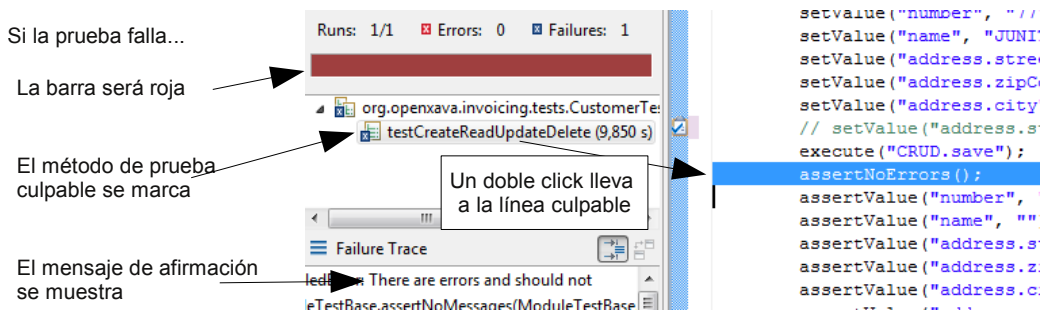


Figura 3.4 Prueba fallida ejecutada desde Eclipse

El *assert* culpable es `assertNoErrors()`, el cual además de fallar muestra en la consola los errores mostrados al usuario. Por eso, en la consola de ejecución de tu prueba verás el mensaje del listado 3.3.

Listado 3.3 Mensaje en consola cuando `assertNoErrors()` falla

```
16-jul-2009 18:03 org.openxava.tests.ModuleTestBase assertNoMessages
SEVERE: Error unexpected: Value for Name in Customer is required
```

El problema es claro. El cliente no se ha grabado porque el nombre es obligatorio, y éste no se ha especificado.

Has aprendido como se comporta la prueba cuando falla. Ahora, puedes descomentar la línea culpable y volver a ejecutar la prueba para verificar que todo sigue en su sitio.

3.3 Añadir el controlador JDBC al *path* de Java

Seguramente te habrás dado cuenta de una horrible traza de error en la consola al ejecutar tu prueba, algo así como la que hay en el listado 3.4.

Listado 3.4 Excepción “JDBC Driver not found” al ejecutar la prueba junit

```
javax.persistence.PersistenceException:
[PersistenceUnit: junit] Unable to build EntityManagerFactory
    at ...
Caused by:
org.hibernate.HibernateException: JDBC Driver class not found:
org.postgresql.Driver
    at ...
```

A pesar de la excepción la prueba funciona bien. OpenXava trata de conectarse a la base de datos para obtener metadatos, si no lo consigue intenta conseguir esos metadatos de otras formas, por eso la prueba funciona.

Vamos a añadir el controlador JDBC de PostgreSQL al *path* de Java de tu proyecto, de esta forma evitaremos esta horripilante traza de error, y aún más importante, tu prueba podrá acceder directamente a la base de datos. Esto es muy importante, porque es un caso común querer usar JPA o JDBC directamente desde una prueba JUnit.

Para añadir el controlador JDBC de PostgreSQL JDBC al *Java Build Path* de tu proyecto sigue las instrucciones de la figura 3.5.

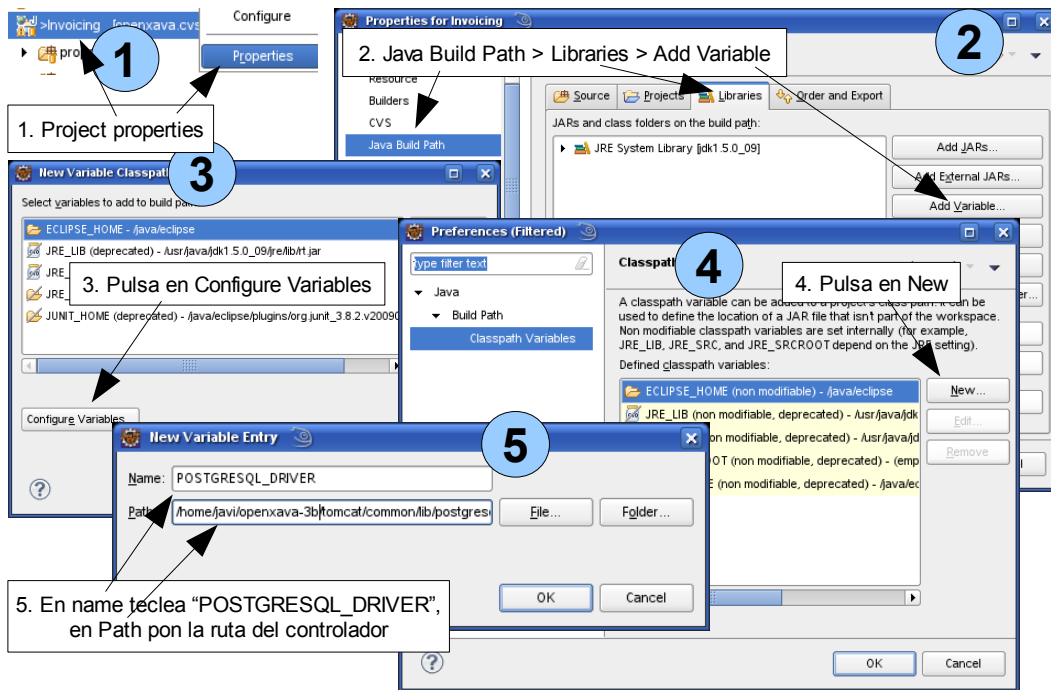


Figura 3.5 Añadir controlador JDBC de PostgreSQL al Java Build Path

Para finalizar cierra todos los diálogos pulsando en OK. En este caso hemos usado una variable *classpath* para añadir la librería al proyecto.

A partir de ahora las pruebas JUnit pueden conectarse a tu base de datos PostgreSQL.

3.4 Crear datos de prueba usando JPA

En tu primera prueba, `CustomerTest`, la prueba misma empieza creando los datos que van a ser usados en el resto de la prueba. Este es un buen enfoque, especialmente si quieres probar la entrada de datos también. Pero a veces te interesa probar solo un pequeño caso que falla, o simplemente tu módulo no permite entrada de datos. En cualquier caso puedes crear los datos que necesites para probar usando JPA desde tu prueba.

3.4.1 Los métodos `setUp()` y `tearDown()`

Vamos a usar `ProductTest` para aprender como usar JPA para crear datos de prueba. Crearemos algunos productos antes de ejecutar cada prueba y los borraremos después. Veamos el código de `ProductTest` en el listado 3.5.

Listado 3.5 Crear y borrar los datos de pruebas para cada caso de prueba

```
package org.openxava.invoicing.tests;

import java.math.*;

import org.openxava.invoicing.model.*;
import org.openxava.tests.*;
import static org.openxava.jpa.XPersistence.*;

public class ProductTest extends ModuleTestBase {

    private Author author; // Declaramos la entidades a crear
    private Category category; // como miembros de instancia para
    private Product product1; // que estén disponibles en todos los métodos de prueba
    private Product product2; // y puedan ser borradas al final de cada prueba

    public ProductTest(String testName) {
        super(testName, "Invoicing", "Product");
    }

    protected void setUp() throws Exception { // setUp() se ejecuta siempre
                                                // antes de cada prueba
        createProducts(); // Crea los datos usados en las pruebas
        super.setUp(); // Es necesario porque ModuleTestBase lo usa para inicializarse
    }

    protected void tearDown() throws Exception { // tearDown() se ejecuta
                                                    // siempre después de cada prueba
        super.tearDown(); // Necesario, ModuleTestBase cierra recursos aquí
        removeProducts(); // Se borran los datos usado para pruebas
    }

    public void testRemoveFromList() throws Exception { ... }

    public void testChangePrice() throws Exception { ... }

    private void createProducts() { ... }

    private void removeProducts() { ... }

}
```

Aquí estamos sobrescribiendo los métodos `setUp()` y `tearDown()`. Estos métodos son métodos de JUnit que son ejecutados justo antes y después de ejecutar cada método de prueba. Creamos los datos de prueba antes de ejecutar cada prueba, y borramos los datos después de cada prueba. Así, cada prueba puede contar con unos datos concretos para ejecutarse. No importa si otras pruebas borran o modifican datos, o el orden de ejecución de las pruebas. Siempre, al principio de cada método de prueba tenemos todos los datos listos para usar.

3.4.2 Crear datos con JPA

El método `createProducts()` es el responsable de crear los datos de prueba

51 Lección 3: Pruebas automáticas

usando JPA. Examinémoslo en el listado 3.6.

Listado 3.6 Crear datos de prueba usando JPA

```
private void createProducts() {  
    // Crear objetos Java  
    author = new Author(); // Se crean objetos de Java convencionales  
    author.setName("JUNIT Author"); // Usamos setters como se suele hacer con Java  
    category = new Category();  
    category.setDescription("JUNIT Category");  
  
    product1 = new Product();  
    product1.setNumber(900000001);  
    product1.setDescription("JUNIT Product 1");  
    product1.setAuthor(author);  
    product1.setCategory(category);  
    product1.setPrice(new BigDecimal("10"));  
  
    product2 = new Product();  
    product2.setNumber(900000002);  
    product2.setDescription("JUNIT Product 2");  
    product2.setAuthor(author);  
    product2.setCategory(category);  
    product2.setPrice(new BigDecimal("20"));  
  
    // Marcar los objetos como persistentes  
    getManager().persist(author); // getManager() es de XPersistence  
    getManager().persist(category); // persist() marca el objeto como persistente  
    getManager().persist(product1); // para que se grabe en la base de datos  
    getManager().persist(product2);  
  
    // Confirma los cambios en la base de datos  
    commit(); // commit() es de XPersistence. Graba todos los objetos en la base de datos  
              // y confirma la transacción  
}
```

Como puedes ver, primero creas los objetos al estilo convencional de Java. Fíjate que los asignamos a miembros de instancia, así puedes usarlos dentro de la prueba. Entonces, los marcas como persistentes, usando el método `persist()` de JPA `EntityManager`. Para obtener el `PersistenceManager` solo has de escribir `getManager()` porque tienes un `import` estático arriba (listado 3.7).

Listado 3.7 Static import para facilitar el uso de `getManager()` y `commit()`

```
import static org.openxava.jpa.XPersistence.*;  
...  
getManager().persist(author);  
    // Gracias al static import de XPersistence es lo mismo que  
XPersistence.getManager().persist(author);  
...  
commit();  
    // Gracias al static import de XPersistence es lo mismo que  
XPersistence.commit();
```

Para finalizar, `commit()` (también de `XPersistence`) graba todos los objetos a la base de datos y entonces confirma la transacción. Después de eso, los datos ya

están en la base de datos listos para ser usados por tu prueba.

3.4.3 Borrar datos con JPA

Después de que se ejecute la prueba borraremos los datos de prueba para dejar la base de datos limpia. Esto se hace en el método `removeProducts()`. Lo puedes ver en el listado 3.8.

Listado 3.8 `removeProducts()/remove()` borran los datos de prueba

```
private void removeProducts() { // Llamado desde tearDown()
    // por tanto ejecutado después de cada prueba
    remove(product1, product2, author, category); // remove() borra
    commit(); // Confirma los datos en la base de datos, en este caso borrando datos
}

private void remove(Object ... entities) { // Usamos argumentos varargs
    for (Object entity: entities) { // Iteramos por todos los argumentos
        getManager().remove(getManager().merge(entity)); // Borrar(1)
    }
}
```

Es un simple bucle por todas las entidades usadas en la prueba, borrándolas. Para borrar una entidad con JPA has de usar el método `remove()`, pero en este caso has de usar el método `merge()` también (1). Esto es porque no puedes borrar una entidad desasociada (*detached entity*). Al usar `commit()` en `createProducts()` todas las entidades grabadas pasaron a ser entidades desasociadas, porque continúan siendo objetos Java válidos pero el contexto persistente (*persistent context*, la unión entre las entidades y la base de datos) se perdió en el `commit()`, por eso tienes que reasociarlas al nuevo contexto persistente. Este concepto es fácil de entender con el código del listado 3.9.

Listado 3.9 Usar `merge()` para reasociar instancias desasociadas

```
getManager().persist(author); // author está asociado al contexto persistente actual
commit(); // El contexto persistente actual se termina, y author pasa a estar desasociado

getManager().remove(author); // Falla porque author está desasociado

author = getManager().merge(author); // Reasocia author al contexto actual
getManager().remove(author); // Funciona
```

A parte de este detalle curioso sobre el `merge()`, el código para borrar es bastante sencillo.

3.4.4 Filtrar datos desde modo lista en una prueba

Ahora que ya sabes como crear y borrar datos para las pruebas, examinemos los métodos de prueba para tu módulo `Product`. El primero es `testRemoveFromList()` que selecciona una fila en el modo lista y pulsa en el

botón “Borrar seleccionados”. Veamos su código en el listado 3.10.

Listado 3.10 Probando el modo lista filtrando los de datos

```
public void testRemoveFromList() throws Exception {
    setConditionValues( // Establece los valores para filtrar los datos
        new String[] { "", "JUNIT" });
    setConditionComparators( // Pone los comparadores para filtrar los datos
        new String[] { "=", "contains_comparator" });
    execute("List.filter"); // Pulsa el botón para filtrar
    assertListRowCount(2); // Verifica que hay 2 filas
    checkRow(1); // Seleccionamos la fila 1 (que resulta ser la segunda)
    execute("CRUD.deleteSelected"); // Pulsa en el botón para borrar
    assertListRowCount(1); // Verifica que ahora solo hay una fila
}
```

Aquí filtramos en modo lista todos los productos que contienen la palabra “JUNIT” (recuerda que has creado dos de estos en el método `createProducts()`), entonces verificamos que hay dos filas, seleccionamos el segundo producto y lo borramos, verificando al final que la lista se queda con un solo producto.

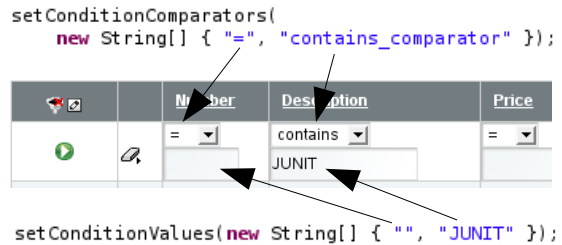


Figura 3.6 Filtrar la lista en una prueba

Has aprendido como seleccionar una fila (usando `checkRow()`) y como verificar el número de filas (usando `assertListRowCount()`). Quizás la parte más intrincada es usar `setConditionValues()` y `setConditionComparators()`. Ambos métodos reciben un *array* de cadenas con valores y comparadores para la condición, tal como muestra la figura 3.6. Los valores son asignados al filtro de la lista secuencialmente (de izquierda a derecha). No es necesario que especifiques todos los valores. El método `setConditionValues()` admite cualquier cadena mientras que `setConditionComparators()` admite los siguientes valores posibles: `starts_comparator`, `contains_comparator`, `=`, `<`, `>`, `>=`, `<=`, `>` y `<`.

3.4.5 Usar instancias de entidad dentro de una prueba

La prueba que queda, `testChangePrice()`, simplemente escoge un producto y cambia su precio. Vamos a usar en él una entidad creada en `createProducts()`. El código está en el listado 3.11.

Listado 3.11 Probar usando un valor de una entidad creada para la prueba

```
public void testChangePrice() throws Exception {
    // Buscar product1
    execute("CRUD.new");
    setValue("number", Integer.toString(product1.getNumber())); // (1)
}
```

```

execute("CRUD.refresh");
assertValue("price", "10.00");

// Cambiar el precio
setValue("price", "12.00");
execute("CRUD.save");
assertNoErrors();
assertValue("price", "");

// Verificar
setValue("number", Integer.toString(product1.getNumber())); // (1)
execute("CRUD.refresh");
assertValue("price", "12.00");
}

```

Lo único nuevo en esta prueba es que para dar valor al número usado para buscar el producto, lo obtenemos de `product1.getNumber()` (1). Recuerda que `product1` es una variable de instancia de la prueba a la que se asigna valor en `createProducts()`, el cual es llamado desde `setUp()`, es decir se ejecuta antes de cada prueba.

Ya tienes la prueba para `Product` y al mismo tiempo has aprendido como probar usando datos de prueba creados mediante JPA.

3.5 Usar datos ya existentes para probar

A veces puedes simplificar la prueba usando una base de datos que contenga los datos necesarios para la prueba. Si no quieres probar la creación de datos desde el módulo, y no borras datos en la prueba, ésta puede ser una buena opción.

Por ejemplo, puedes probar `Author` y `Category` con una prueba tan simple como la del listado 3.12.

Listado 3.12 Probar confiando en datos ya existentes en la base de datos

```

public class AuthorTest extends ModuleTestBase {

    public AuthorTest(String testName) {
        super(testName, "Invoicing", "Author");
    }

    public void testReadAuthor() throws Exception {
        assertValueInList(0, 0, "JAVIER CORCOBADO"); // El primer Author en la
                                                    // lista es JAVIER CORCOBADO
        execute("Mode.detailAndFirst"); // Al cambiar a modo detalle
                                        // se visualiza el primero objeto de la lista
        assertValue("name", "JAVIER CORCOBADO");
        assertCollectionRowCount("products", 2); // Tiene 2 productos
        assertValueInCollection("products", 0, // Fila 0 de products
                                "number", "2"); // tiene "2" en la columna "number"
        assertValueInCollection("products", 0,

```

```

        "description", "Arco iris de lágrimas");
    assertEquals("products", 1, "number", "3");
    assertEquals("products", 1,
        "description", "Ritmo de sangre");
    }
}

```

Esta prueba verifica que el primer autor en la lista es “JAVIER CORCOBADO”, entonces va al detalle y confirma que tiene una colección llamada *products* con 2 productos: “Arco iris de lágrimas” y “Ritmo de sangre”. De paso, has aprendido como usar los métodos `assertValueInList()`, `assertValueInCollection()` y `assertCollectionRowCount()`.

Podemos usar la misma técnica para probar el módulo `Category`. Veámoslo en el listado 3.13.

Listado 3.13 Probar sólo el modo lista con datos existentes

```

public class CategoryTest extends ModuleTestBase {

    public CategoryTest(String testName) {
        super(testName, "Invoicing", "Category");
    }

    public void testCategoriesInList() throws Exception {
        assertEquals(0, 0, "MUSIC"); // Fila 0 columna 0 contiene "MUSIC"
        assertEquals(1, 0, "BOOKS"); // Fila 1 columna 0 contiene "BOOKS"
        assertEquals(2, 0, "SOFTWARE"); // Fila 2 columna 0 contiene "SOFTWARE"
    }
}

```

En este caso solo verificamos que en la lista las tres primeras categorías son “MUSIC”, “BOOKS” y “SOFTWARE”.

Puedes ver como la técnica de usar datos preexistentes de una base de datos de prueba te permite crear pruebas más simples. Empezar con una prueba simple e ir complicándolo bajo demanda es una buena idea.

3.6 Probar colecciones

Es el momento de enfrentarnos a la prueba del módulo principal de tu aplicación, `InvoiceTest`. Por ahora la funcionalidad del módulo `Invoice` es limitada, solo puedes añadir, borrar y modificar facturas. Aun así, esta es la prueba más extensa; además contiene una colección, por tanto aprenderás como probar las colecciones.

3.6.1 Dividir la prueba en varios métodos

El listado 3.14 muestra el código de InvoiceTest.

Listado 3.14 La prueba para crear una factura está dividida en varios métodos

```
package org.openxava.invoicing.tests;

import java.text.*;
import java.util.*;
import javax.persistence.*;
import org.openxava.tests.*;
import org.openxava.util.*;

import static org.openxava.jpa.XPersistence.*; // Para usar JPA

public class InvoiceTest extends ModuleTestBase {

    private String number; // Para almacenar el número de la factura que probamos

    public InvoiceTest(String testName) {
        super(testName, "Invoicing", "Invoice");
    }

    public void testCreateInvoice() throws Exception { // El método de prueba
        verifyDefaultValues();
        chooseCustomer();
        addDetails();
        setOtherProperties();
        save();
        verifyCreated();
        remove();
    }

    private void verifyDefaultValues() throws Exception { ... }

    private void chooseCustomer() throws Exception { ... }

    private void addDetails() throws Exception { ... }

    private void setOtherProperties() throws Exception { ... }

    private void save() throws Exception { ... }

    private void verifyCreated() throws Exception { ... }

    private void remove() throws Exception { ... }

    private String getCurrentYear() { ... }

    private String getCurrentDate() { ... }

    private String getNumber() { ... }

}
```

El único método de prueba de esta clase es testCreate(), pero dado que es bastante extenso, es mejor dividirlo en varios métodos más pequeños. De hecho,

es una buena práctica OO⁵ escribir métodos cortos.

Ya que el método es corto puedes ver con un solo golpe de vista que es lo que hace. En este caso verifica los valores por defecto para una factura nueva, escoge un cliente, añade las líneas de detalle, añade otras propiedades, graba la factura, verifica que ha sido guardada correctamente y al final la borra. Entremos en los detalles de cada uno de estos pasos.

3.6.2 Verificar valores por defecto

Lo primero es verificar que los valores por defecto para una factura nueva son calculados correctamente. Esto se hace en el método `verifyDefaultValues()`. Está en el listado 3.15.

Listado 3.15 Verificar los valores por defecto al crear una nueva factura

```
private void verifyDefaultValues() throws Exception {
    execute("CRUD.new");
    assertValue("year", getCurrentYear());
    assertValue("number", getNumber());
    assertValue("date", getCurrentDate());
}
```

Cuando el usuario pulsa en “Nuevo”, los campos año, número y fecha tienen que rellenarse con datos válidos. El método `verifyDefaultValues()` precisamente comprueba esto. Usa varios métodos de utilidad para calcular los valores esperados. Los puedes ver en el listado 3.16.

Listado 3.16 Métodos de utilidad usados por `verifyDefaultValues()`

```
private String getCurrentYear() { // Año actual en formato cadena
    return new SimpleDateFormat("yyyy").format(new Date()); // La forma típica
                                                             // de hacerlo con Java
}

private String getCurrentDate() { // Fecha actual como una cadena en formato corto
    return DateFormat.getDateInstance( // La forma típica de hacerlo con Java
        DateFormat.SHORT).format(new Date());
}

private String getNumber() { // El número de factura para una factura nueva
    if (number == null) { // Usamos inicialización vaga
        Query query = getManager(). // Una consulta JPA para obtener el último número
            createQuery(
                "select max(i.number) from Invoice i where i.year = :year"
            );
        query.setParameter("year",
            Dates.getYear(new Date())); // Dates es una utilidad de OpenXava
        Integer lastNumber = (Integer) query.getSingleResult();
        if (lastNumber == null) lastNumber = 0;
        number = Integer.toString(lastNumber + 1); // Añadimos 1 al
                                                    // último número de factura
    }
}
```

```

    }
    return number;
}

```

Los métodos `The getCurrentYear()` y `getCurrentDate()` usan técnicas clásicas de Java para formatear la fecha como una cadena.

El método `getNumber()` es un poco más complejo: usa JPA para calcular el último número de factura del año en curso y después devuelve este valor más uno. Dado que acceder a la base de datos es más pesado que un simple cálculo Java, usamos una inicialización vaga. Una inicialización vaga retrasa el cálculo hasta la primera vez que se necesita, y después lo almacena para futuros usos. Esto lo hacemos guardando el valor en el campo `number`.

Fíjate en el uso de la clase `Dates` para extraer el año de la fecha. `Dates` es una clase de utilidad que puedes encontrar en `org.openxava.util`.

3.6.3 Entrada de datos

Ahora es el momento de `chooseCustomer()` de la factura. Mira el código en listado 3.17.

Listado 3.17 Probar escogiendo un cliente para una factura

```

private void chooseCustomer() throws Exception {
    setValue("customer.number", "1");
    assertValue("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");
}

```

Al introducir el número de cliente el nombre del cliente se rellena con un valor apropiado. Con esto asociamos el cliente 1 con la factura actual.

Y ahora viene la parte más peliaguda de la prueba: añadir las líneas de detalle. Tienes el código de `addDetails()` en el listado 3.18.

Listado 3.18 Probar añadiendo elementos a la colección

```

private void addDetails() throws Exception {
    // Añadir una línea de detalle
    assertCollectionRowCount("details", 0); // La colección esta vacía
    execute("Collection.new", // Pulsa en el botón para añadir un nuevo elemento
           "viewObject=xava_view_details"); // viewObject es necesario para determinar
                                           // a que colección nos referimos

    setValue("product.number", "1");
    assertValue("product.description",
               "Peopleware: Productive Projects and Teams");
    setValue("quantity", "2");
    execute("Collection.saveAndStay"); // Graba el elemento de la colección
                                     // sin cerrar el diálogo

    assertNoErrors(); // No hay errores al grabar el detalle

    // Añadir otro detalle
}

```


59 Lección 3: Pruebas automáticas

```
setValue("product.number", "2");
assertValue("product.description", "Arco iris de lágrimas");
setValue("quantity", "1");
execute("Collection.save"); // Graba el elemento de la colección cerrando el diálogo
assertNoErrors();
assertCollectionRowCount("details", 2); // Ahora tenemos 2 filas
}
```

Probar una colección es exactamente igual que probar cualquier otra parte de tu aplicación, solo has de seguir los mismos pasos que un usuario haría con el navegador. Nota que has de usar `viewObject=xava_view_details` como argumento para algunas acciones de la colección.

Ahora que tenemos los detalles añadidos, vamos a llenar los datos restantes y grabar la factura. Los datos restantes se establecen en el método `setOtherProperties()` (listado 3.19).

Listado 3.19 Poner valor a las propiedades restantes antes de grabar la factura

```
private void setOtherProperties() throws Exception {
    setValue("remarks", "This is a JUNIT test");
}
```

Aquí ponemos valor al campo `remarks`. Y ahora estamos listos para grabar la factura. Lo puedes ver en listado 3.20.

Listado 3.20 Grabar la factura desde la prueba JUnit

```
private void save() throws Exception {
    execute("CRUD.save");
    assertNoErrors();

    assertValue("customer.number", "");
    assertCollectionRowCount("details", 0);
    assertValue("remarks", "");
}
```

Simplemente pulsa en “Save”, entonces verifica que no ha habido errores y la vista se ha limpiado.

3.6.4 Verificar los datos

Ahora, buscamos la factura recién creada para verificar que ha sido grabada correctamente. Esto se hace en el método `verifyCreated()` que puedes ver en listado 3.21.

Listado 3.21 Verificar que los datos de la factura han sido grabado correctamente

```
private void verifyCreated() throws Exception {
    setValue("year", getFullYear()); // El año actual en el campo año
    setValue("number", getNumber()); // El número de la factura usada en la prueba
    execute("CRUD.refresh"); // Carga la factura desde la DB
}
```

```

// En el resto de la prueba confirmamos que los valores son los correctos
assertValue("year", getFullYear());
assertValue("number", getNumber());
assertValue("date", getFullYear());

assertValue("customer.number", "1");
assertValue("customer.name", "FRANCISCO JAVIER PANIZA LUCAS");

assertCollectionRowCount("details", 2);

// Fila 0
assertValueInCollection("details", 0, "product.number", "1");
assertValueInCollection("details", 0, "product.description",
    "Peopleware: Productive Projects and Teams");
assertValueInCollection("details", 0, "quantity", "2");

// Fila 1
assertValueInCollection("details", 1, "product.number", "2");
assertValueInCollection("details", 1, "product.description",
    "Arco iris de lágrimas");
assertValueInCollection("details", 1, "quantity", "1");

assertValue("remarks", "This is a JUNIT test");
}

```

Después de buscar la factura creada verificamos que los valores que hemos grabado están ahí. Si la prueba llega a este punto tu módulo Invoice funciona bien. Solo nos queda borrar la factura creada para que la prueba se pueda ejecutar la siguiente vez. Hacemos esto en el método `remove()` (listado 3.22).

Listado 3.22 Borrar la factura usada en la prueba

```

private void remove() throws Exception {
    execute("CRUD.delete");
    assertNoErrors();
}

```

Simplemente presiona en “Borrar” y verifica no se han producido errores.

¡Enhorabuena! Has completado tu InvoiceTest.

3.7 Suite

Tienes 5 casos de prueba que velan por tu código, preservando la calidad de tu aplicación. Cuando termines alguna mejora o corrección en tu aplicación ejecuta todas tus pruebas unitarias para verificar que la funcionalidad existente no se ha roto.

Tradicionalmente, para ejecutar todas las pruebas de tu aplicación deberías crear una suite de pruebas, y ejecutarla. Una suite de pruebas es una clase que agrega todas tus pruebas JUnit para que puedas ejecutarlas todas de un golpe.

61 Lección 3: Pruebas automáticas

Afortunadamente, si trabajas con Eclipse no necesitas escribir una clase de suite, Eclipse te permite ejecutar todas las pruebas de tu aplicación automáticamente, como muestra la figura 3.7.

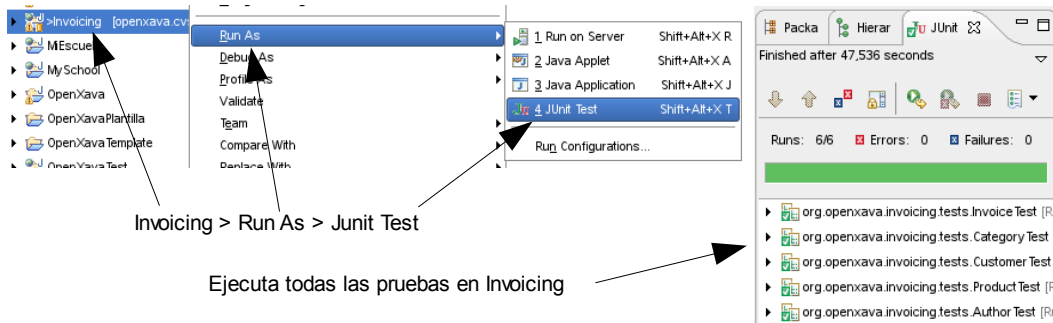


Figura 3.7 Ejecuta todas las prueba del proyecto a la vez

Es decir, si ejecutas *Run As > JUnit Test* en el proyecto, se ejecutarán todas sus pruebas JUnit.

3.8 Resumen

Has automatizado las pruebas de toda la funcionalidad actual de tu aplicación. Puede parecer que este código de prueba es mucho más largo y aburrido que el código real de la aplicación. Pero recuerda, el código de prueba es el tesoro más valioso que tienes. Quizás ahora no me creas, pero trata de hacer pruebas JUnit y una vez te hayan salvado la vida, ya no podrás desarrollar sin pruebas automáticas nunca más.

¿Qué probar? No hagas pruebas exhaustivas al principio. Es mejor probar poco que no probar nada. Si tratas de hacer pruebas muy exhaustivas acabarás no haciendo pruebas en absoluto. Empieza haciendo algunas pruebas JUnit para tu código, y con cada nueva característica o nuevo arreglo añade nuevas pruebas. Al final, tendrás una suite de pruebas muy completa. En resumen, prueba poco, pero prueba siempre.

Sí, hacer pruebas automáticas es una tarea continua. Y para predicar con el ejemplo a partir de ahora escribiremos todas las pruebas para el código que desarrollemos en el resto del libro. De esta manera aprenderás más trucos sobre las pruebas JUnit en las siguientes lecciones.